ICASE

THE PROVISION OF RECOVERABLE INTERFACES

T. Anderson

and

P. A. Lee

Report Number 79-9

April 16, 1979

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia

Operated by the

UNIVERSITIES SPACE USRA RESEARCH ASSOCIATION

The Provision of Recoverable Interfaces

T. Anderson†
Institute for Computer Applications in Science and Engineering

and

P. A. Lee
Computing Laboratory, The University,
Newcastle upon Tyne, U. K.

Abstract

The recovery block scheme has been proposed as one method of providing fault tolerant software, and is dependent on the availability of recoverable interfaces so that any damage caused by an erroneous program can be repaired by backward error recovery. However, it is clear that the interface provided by the hardware in any practical system will contain unrecoverable objects. This paper investigates a method of structuring a system into multiple levels so that a level of software can "hide" the unrecoverable features of an interface and provide a new interface with recoverable objects to programs needing facilities for backward error recovery. The paper discusses this organization of recovery in such a system.

† On leave from the Computing Laboratory, The University, Newcastle upon Tyne, U. K.

The research for the first author was partially supported under NASA Contracts NAS1-14101 and NAS1-14472 while he was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

1. Introduction

In order to attain a high level of reliability the designer of a system will attempt to ensure first that the system does not contain faults, and second that those faults which it does contain (since the first objective will not be achieved) are tolerated and do not cause the system to fail. An important element in any measures for fault tolerance is a means of error recovery, that is of transforming a state of the system which (due to some fault) is erroneous to a state from which the system can continue to provide its specified service.

Many of the erroneous states which can occur in the operation of a system can be anticipated. In consequence it may well be possible to construct specific error recovery measures to rectify such errors. Indeed, most of the work on tolerance for faults in the hardware of computer systems has catered only for predicted error situations caused by (anticipated) component failures. Techniques for coping with component failure can be embodied in the hardware itself [Avi75], or in the software of a system in the form of exception handling routines [Goo75]. However, faults in the design of a system can lead to erroneous states which are unanticipated and cannot be predicted. Of course, faults in software are always due to deficiencies in design and in consequence the techniques which have been quite effective in averting system failures due to hardware faults are inadequate and inappropriate as a defence against software faults.

Any technique for providing recovery in an unanticipated situation must be of a very general nature and should not place undue reliance on an erroneous state caused by a design fault. One such technique is to abandon the erroneous state and restore the objects in the system to the values which pertained in some prior state. This approach has been termed "backward error recovery" [Ran78] since an earlier state is restored and some system activity is in consequence abandoned. Backward error recovery is an important technique, for if it is employed in a system then recovery can be obtained from the effects of a wide class of faults, including those of design. Successful restoration of a prior state ensures the elimination of all errors generated by any fault which occurred after that earlier state; thus a powerful fault tolerance capability can be provided.

Consider, for example, the interface between the "hardware" and "software" of a computer system. The hardware machine interprets the machine language programs comprising the software of the system, and provides various abstract objects such as registers, words of memory in main storage, pages of data on disc, and I/O devices. One of the aspects of providing fault tolerance at the software level is to provide backward error recovery for the objects manipulated by the programs. (Objects for which backward error recovery is provided will be termed "recoverable" in this paper.) The recovery cache has been proposed [Hor74] as a mechanism for providing, by hardware, recovery for those objects that reside in the main store of the machine, and it has been demonstrated that this is a feasible and efficient technique [And76, Shr78, Lee79]. However, the optimised checkpointing strategy employed by the recovery cache is less appropriate for the provision of backward error recovery for most of the other objects supported by the hardware interpreter, particularly for those objects which interact with the external environment of the system.

This paper describes an approach to the construction of complex systems which involves structuring a system into a hierarchy of interfaces or levels such that a higher level can provide recoverable abstract objects which it

implements from the (relatively) concrete objects available from a lower level. Examples of systems in which a hierarchical multi-level approach has been adopted have been described in the literature [Dij68,Lis72]; a detailed examination of a model of recovery in multi-level systems has also been published [And78]. This paper considers the way in which a multi-level recoverable system could be designed and illustrates the approach by means of a simple example. Some useful observations on the practical details of the approach are made.

2. Basic Recovery Concepts

First, consider the simple case of a program running on a given interface L. The term recoverability is, as indicated above, taken to mean the ability to recover an earlier state of the objects available on an interface, thereby undoing the effects of operations that were performed on those objects. To provide such backward error recovery necessitates the recording of recovery data which can be used for this state restoration. Correspondingly, programs which manipulate the recovery data are referred to as recovery programs.

In the rest of this paper it will be assumed that the following basic recovery features are available to programs executed on an interface:

(i) The interface provides both recoverable and unrecoverable objects.

(ii) A program can establish recovery points which ensure that the current state of the recoverable objects of the interface is (at least conceptually) recorded as recovery data.

(iii) Recovery points can be discarded with the effect that recovery data maintained for recovery to those recovery points is discarded.

A recovery point is said to be active from when it is established until it is discarded. The term recovery region will be used to refer to the period for which a recovery point is active (figure 1 shows two nested recovery regions).

3. Multi-Level Systems

A systematic method of designing a complex computer system is to adopt a hierarchical approach: starting from a given hardware interface L0, a first layer of software is added to obtain a more attractive interface L1; this process is repeated to obtain L2, and so on. The resulting system is termed multi-level in that a number of interfaces, or levels, can be discerned in its implementation.

The most powerful and general method of providing a new interface is to use interpretation techniques. For example, a new level L1 can be constructed from an existing hardware interface L0 by providing a software-implemented interpreter I1 (which is, of course, executed on L0). This is depicted in figure 2.

The characteristic feature of this approach is that an interpreter has complete responsibility for the support of the new interface - every operation performed by a program I2 on objects available on L1 in figure 2 is directly supported by I1. The design of a multi-level system can be simplified by the adoption of interpretation techniques because the implementation of each of the levels supported by interpretation is completely independent of the implementation of the underlying levels. For example, in figure 2 the recoverability of the objects available on L0 has no direct bearing on the

recoverability of objects available on L1. Any recovery features available on L1 have to be explicitly provided by I1.

The major practical disadvantage with interpretation is in the substantial overhead which it incurs. Indeed, if it is desired that a new interface L1 is to have many features in common with the underlying interface L0 then interpretation can be a costly technique to utilise. There is an alternative to full interpretation: if the hardware machine makes available sufficiently powerful extension facilities then it may not be necessary to support new interfaces by further levels of interpretation; instead, these may be provided by extending the hardware provided features. Figure 3 illustrates a computer system in which the hardware-provided interface L0 (itself supported by a hardware-implemented interpreter I0) provides extension facilities. Each new interface Li (i = 1,2) is constructed as an extension of Li-1, the extension being implemented by means of a program Ei which is executed on the interface Li-1.

Each program Ei is referred to as an interpreter extension. Every interaction with the interface L2 is first examined by the underlying interpreter I0, which determines whether that interaction is directly supported by I0 itself or, if not, which of the interpreter extensions (E1,E2) does support that interaction. Thus, the interactions of a program may be supported by any lower extensions or by I0. The layout of figure 3 is intended to indicate that interfaces L0, L1 and L2 have many behavioural properties in common.

Example:

> The nucleus of most operating systems can be regarded as an interpreter extension which provides a user interface from the underlying hardware interface by removing certain privileged instructions and adding a set of 'operating system call' instructions.

The potential advantage to be gained from using an interpreter extension to implement a new interface (when this is possible) in preference to a further level of interpretation is in avoiding the overhead that the latter entails. It should be noted that to a program being executed on an interface, it is completely immaterial whether that interface is implemented by an interpreter extension or not.

As discussed above, the advantages of full interpretation in the implementation of multi-level systems may be diminished by the overheads incurred, and it is likely that many practical multi-level systems will be constructed using interpreter extensions. Thus the rest of this paper concentrates on the provision of recoverable interfaces by means of a hierarchy of interpreter extensions. In order to illustrate the discussion a simple example multi-level system will be presented.

4. A Simple File System

The example system in which the provision of recoverable interfaces is to be considered is a rudimentary filing system; it supports only a single file for use by a single user. The implementation of the system is as depicted in figure 3, and has the following characteristics.

L0: Among the objects available on L0 are variables held in main memory, and disc pages held on secondary storage. The disc is accessed by means of the operations 'readdisc' and 'writedisc'. These objects and the operations to manipulate them are supported by the underlying interpreter I0.

L1: The interpreter extension E1 extends L0 by providing operations to acquire and release disc pages (operations 'getdiscpage' and 'releasediscpage'), maintaining a list of the free disc pages in main memory.

L2: The second interpreter extension E2 prevents the user program P from directly accessing the disc pages. Instead, P is given access to a file; the user views this file as an indexed sequence of lines of text, with operations 'openfile', 'closefile', 'readline' and 'writeline'. The concrete representation maintained by E2 for the file consists of a set of the unrecoverable disc pages, a copy (called 'pagebuffer') of the most recently accessed disc page, and an array of disc page addresses (called 'filemap'). The objects pagebuffer and filemap are held in main memory. Each entry in filemap points to one of the disc pages currently representing the file. When P accesses the file, either to read or write a line, the access is actually applied (by E2) to pagebuffer. If the line in question is not present, because pagebuffer is empty or contains the wrong disc page, then the relevant disc page is copied into pagebuffer (if pagebuffer contains an updated disc page then this must first be copied back to the disc).

5. Provision of Recovery

If backward error recovery is to be provided to a program in a system with interpreter extensions, then whenever the program manipulates a recoverable object within a recovery region, it must be ensured that the necessary recovery data is recorded. If the recoverable object is supported by the underlying interpreter then the recovery data will be maintained by the interpreter. Similarly, an interpreter extension may need to record recovery data so that it can provide recovery to any recoverable objects it supports.

It will be assumed that the underlying interpreter I0 of the file system example provides recovery for variables in main memory, but not for disc pages (that is, words of main memory are recoverable and disc pages are unrecoverable). Although the first extension (E1) provides no recovery features, the second extension (E2) is intended to provide a recoverable file; since the file is implemented on disc as well as in main storage, E2 will have to include recovery programs and data, as is depicted in figure 4.

In order that an extension, such as E2 in figure 4, can perform the necessary actions for recovery, the underlying interpreter must invoke the extension whenever a program using that extension establishes or discards a recovery point as well as whenever recovery is required. When recovery is required for P (the user of the file system) then the extension E2 must restore the prior state of the file and the interpreter I0 must restore those variables of P which are held in main memory.

The basis of one method by which E2 could provide recovery for the file is as follows: whenever P establishes a recovery point, E2 must ensure that the disc pages which represent the file are not subsequently overwritten. When pagebuffer is to be copied back to the disc, instead of overwriting the original disc page, an unused disc page is acquired and pagebuffer is written to this new page. Clearly, the appropriate entry in filemap must be changed to point to the new disc page, and in consequence the disc address of the old disc page must be recorded as recovery data by E2.

An interpreter extension can itself make use of recoverable objects, either in conjunction with its own use of recovery points or simply for convenience in representing objects maintained by the extension.

In the file system, the objects filemap and pagebuffer used by E2 are recoverable since they reside in main storage. The question which will now be considered is whether recoverable objects used by an interpreter extension should be restored by the underlying interpreter when recovery is provided to a program which has called the extension. For example, should recovery of P cause the objects filemap and pagebuffer to be restored by I0?

6. Disjoint and Inclusive Recovery

The distinction between the two recovery schemes discussed below stems from the way in which an interpreter extension is regarded as fitting into the structure of the system. As its name suggests, an interpreter extension is an extension of an underlying interpreter and could therefore be regarded as being a part of that interpreter (at least conceptually), and hence independent (or disjoint) from any calling program. If the extension and calling program are regarded as disjoint components of the system it seems legitimate that recovery for one should not imply that any recovery is required for the other. In consequence an extension would be wholly responsible for the recovery of objects it was maintaining. A scheme of recovery for multi-level systems having these characteristics is termed a disjoint recovery scheme.

If disjoint recovery is adopted for the file system then the objects filemap and pagebuffer would not be restored to their prior states by I0 when recovery is provided to P. E2 must still be able to restore the prior state of the file - but this is easily achieved. The recovery data recorded by E2 simply needs to indicate which filemap entries must be restored and the disc addresses to which they should be reset (that is, the address of the old disc pages discussed in the previous section). Using this information, E2 can reset filemap to its state at the time the recovery point was established and can also release the new disc pages that had been acquired. Note that pagebuffer need not be restored. The recovery program of E2 merely empties pagebuffer since any subsequent access of the file by P will result in a disc page being copied into pagebuffer. All of these actions ensure that the file is restored to the abstract state which existed when P established the recovery point. This example also illustrates that provision of the abstraction of recovery for an object does not imply that an exact prior state of that object must be restored; there may be many concrete states which have the same abstract state. The disjoint recovery scheme can take advantage of this, as illustrated here, when providing recovery. (A more detailed elaboration of the file system program of E2 with disjoint recovery is supplied in the Appendix.)

There is a second way in which an extension may be regarded as fitting into the structure of the system. Instead of taking an extension to be disjoint from a calling program an alternative is to regard the calling program as being inclusive of the extension, the extension then being regarded as a nested component of the calling program. It then seems natural that recovery of the calling program should also include recovery of the extension. In consequence an extension would only need to record and maintain recovery data relating to the use of any unrecoverable objects it manipulates on behalf of a calling program; recovery of any recoverable objects used by the extension would be automatically provided by lower extensions or by the underlying interpreter. A scheme of recovery for multi-level systems having these characteristics is termed an inclusive recovery scheme.

If inclusive recovery is adopted for the file system then when recovery is invoked for P, the prior states of filemap and pagebuffer will be

automatically restored by IO. As with the disjoint scheme, E2 must acquire new disc pages to avoid overwriting the disc pages which represented the file at the time the recovery point was established. Automatic restoration of filemap thus ensures that the file is restored to its prior state, and the recovery program of E2 merely has to release the newly acquired disc pages, the addresses of which would have been recorded as recovery data (see the Appendix for a more detailed description).

One complication which arises with inclusive recovery concerns the recovery data maintained by the extension (such as the addresses of newly acquired disc pages in the file system). If this data is retained in recoverable objects (perhaps to enhance the recovery capabilities of the extension itself) then this information would be lost if the objects were restored by the underlying interpreter before the recovery program of the extension was executed. This difficulty can be avoided in a number of ways. The simplest, but least acceptable, approach is to stipulate that the recovery data of an extension must be maintained in unrecoverable objects; unfortunately the recovery programs of the extension are then unable to derive any local benefit from the recovery capability of the underlying interpreter. The most general solution is to allow an extension to specify that the provisions of disjoint recovery should be applied to the objects it uses to hold recovery data. The code presented in the Appendix assumes that this strategy is being employed. Alternatively, it may be possible to ensure that the recovery programs of an extension are executed before recovery is provided by the underlying interpreter. In a multi-level system this implies that when recovery is required for a program, the underlying interpreter must invoke the recovery programs of all relevant extensions in order from right to left, that is from "most" to "least" abstract. (The adoption of this strategy would also allow minor optimisations of the code presented in the Appendix to be made.)

7. General Comments on the Recovery Schemes in the Example

At a superficial level, there would appear to be only minor differences between the implementation of recovery in the example system utilising disjoint or inclusive recovery. Certainly, it would be wrong to try to draw firm conclusions about the usefulness of either scheme based on this simple example. It is claimed that both schemes provide exactly the same abstraction of recovery to the user of the file (P) and, as expected, it is necessary to investigate the implementation of this abstraction in E2 to distinguish the schemes.

As far as recovery is concerned, the implementation of the disjoint scheme naturally has to do more work than that of the inclusive scheme. However, in this example the extra work turns out to be relatively minor because, since the disjoint scheme gives full control over recovery, it is possible to adopt a reasonably efficient method of restoring a concrete state of the file in order to provide the abstraction of recovery. Thus, for example, pagebuffer does not need to be restored to the state that pertained when P established a recovery point. In contrast, with the inclusive scheme, pagebuffer is restored automatically without cost (strictly, at the cost of recovery data in the lower machine). The disjoint scheme does incur the cost of the disc access to reset pagebuffer when the file is next accessed by P. It should also be noted that if the 'getdiscpage' and 'releasediscpage' procedures of E1 were made recoverable by E1 then there would be no need whatsoever for a recovery program in E2 with the inclusive recovery scheme. However, the recoverability provided by E1 would have no impact on the recovery program of E2 in the disjoint scheme.

On the other hand, the disjoint scheme has to checkpoint information maintained in main memory about the file status when a recovery point is established. In the current example there is minimal information; a more practical file system which maintained other state information (time, data altered, owner, size, ...) would necessitate the recording of additional information as recovery data of the disjoint scheme.

The example also highlights the problems of recording recovery data in recoverable objects with the inclusive recovery scheme. Anderson, Lee and Shrivastava [And78] stated that the natural order for recovery of the extensions is from least to most abstract. However, it has been shown that providing recovery in the reverse order would provide the desired effect and would simplify the system in that the objects used to hold recovery data would not need to be specially identified and dealt with. Indeed, the recording of recovery data may be simplified by the adoption of this strategy.

It is likely that extensions themselves contain faults. It should be noted that neither recovery scheme precludes an extension from establishing its own local recovery points as part of its fault tolerance strategies. Indeed, this may be considered necessary to provide reliable extension operation, although the simple example presented here contains no such strategies.

8. Summary

By discussing the details of the provision of backward error recovery in a very simple file system, the salient characteristics of two schemes of recovery in multi-level systems have been presented. The disjoint scheme gives complete control over recovery to an extension but only at the price of having to re-implement recovery when that provided by the underlying interpreter could have been adequate. The inclusive scheme enables an extension to take advantage of the recovery provided by the underlying interpreter in providing its own recovery capability, but is complicated by the need to obtain disjoint recovery for its recovery data. A practical solution to this problem has been discussed. The example system also illustrates how unrecoverable features of a low level interface can be eliminated by replacing them with recoverable abstract objects in a new interface. A reasonably detailed elaboration of the file system program of E2 is attached as an Appendix.

Acknowledgements

We would like to thank our fellow members of the Reliability Project, which is sponsored by the Science Research Council of Great Britain, at the University of Newcastle upon Tyne.

References

[And76] T. Anderson and R. Kerr, "Recovery blocks in action". Proc. 2nd Int. Conf. on Software Engineering, San Francisco, Oct. 1976, pp.447-457.

[And78] T. Anderson, P.A. Lee and S.K. Shrivastava, "A model of recoverability in multilevel systems". IEEE Trans. on Software Eng., Vol. SE-4, Nov. 1978, pp. 486-494.

[Avi75] A. Avizienis, "Architecture of fault-tolerant computing systems". Proc. of 5th IEEE Int. Symp. on Fault-tolerant Computing, Paris, June 1975, pp. 3-16.

[Dij68] E.W. Dijkstra, "The structure of the "THE" multiprogramming system". Commun. Ass. Comput. Mach., Vol. 11, May 1968, pp. 341-346.

[Goo75] J.B. Goodenough, "Exception handling: issues and a proposed notation". Commun. Ass. Comput. Mach., Vol. 18, Dec. 1975, pp.683-696.

[Hor74] J.J. Horning et al., "A program structure for error detection and recovery". Lecture Notes in Computer Science 16, Springer, Berlin, 1974, pp. 177-193.

[Lee79] P.A. Lee, N.Ghani and K. Heron, "A recovery cache for the PDP-11". Digest of papers, FTCS-9, Madison, June 1979.

[Lis72] B.H. Liskov, "The design of the Venus operating system". Commun. Ass. Comput. Mach., Vol. 15, March 1972, pp. 144-149.

[Shr78] S.K. Shrivastava and A.A. Akinpelu, "Fault-tolerant sequential programming using recovery blocks". Digest of papers, FTCS-8, Toulouse, June 1978, p. 207.

Appendix

The code which follows is written in a Pascal-like language. The procedures which are made available to the user program P (as operations) are distinguished by the keyword entry. To make it clear to the reader which operations are provided by the extension E1 and which by the interpreter I0, invocation of such operations will be prefixed by "E1." and "I0." respectively. The establishrp, discardrp and recover procedures are those which it is assumed are automatically invoked by I0 as noted in the paper.

The declarations of a number of procedures have been omitted for brevity. The tasks performed by procedures readline, convert, put-in-cache and extract-from-cache should be clear from their names and invocations. Procedures initialise-cache and tidy-up-cache are merely to allow for any necessary updating of the housekeepingvars of the cache. Procedure cacheing-required determines whether recovery data must be entered in the recovery cache, according to whether the disc page which would have been overwritten is one which represented a part of the file at the time the recovery point was established.

As a further simplification, recovery in the example is only considered for a single recovery point. The elementary modifications necessary to cope with multiple nested recovery regions are left as an exercise for the interested reader.

Code is presented for E2 with disjoint recovery, and then for E2 with inclusive recovery.

DISJOINT RECOVERY IN E2

```
constant filemaplocation =...; "address for disc copy of filemap"
type filemapindex = (1..N);
     discaddress  = ...;
     cacheentry   = record oldpage  : discaddress;
                           index : filemapindex
                    endrecord;
     line = ...;

var filemap : array [filemapindex] of discaddress;
    pagebuffer : array [1..M] of line;
    activepageno : (0..N);  "filemapindex of discpage
                             currently in pagebuffer"
    writtento : boolean;
    status : (open, closed) initially closed;
    cache : record housekeepingvars : ...;
                   oldfilestatus : (open,closed);
                   region : array [1..P] of cacheentry;
            endrecord;
```

```
entry procedure openfile;
begin if status = open then signalerror else
  begin
  IO.readdisc(filemaplocation,filemap); "read filemap into core"
  activepageno:=0; writtento:=false; status:=open;
  end;
end openfile;

entry procedure closefile;
begin if status = closed then signalerror else
  begin if writtento then copybackpagebuffer;
  IO.writedisc(filemaplocation,filemap); "assume written to"
  status:=closed;
  end;
end closefile;

entry procedure writeline(lineno:integer,linecontents:line);
var pageno : filemapindex; displacement : integer;
begin if status = closed then signalerror else
  begin convert(lineno,pageno,displacement);
  getpage(pageno);
  pagebuffer[displacement]:=linecontents;
  writtento:=true;
  end;
end writeline;

procedure getpage(pagenumber : filemapindex);
begin if pagenumber = activepageno then return;
if writtento then copybackpagebuffer;
IO.readdisc(filemap[activepageno],pagebuffer);
activepageno:=pagenumber;
writtento:=false;
end getpage;

procedure copybackpagebuffer;
var newpage : discaddress;
begin if cacheing-required then "record recovery data"
  begin newpage:=E1.getdiscpage;
  "cache old disc page address and its filemap index"
  put-in-cache(filemap[index],index);
  filemap[index]:=newpage; "reset filemap for new page"
  end;
IO.writedisc(filemap[index],pagebuffer);
end copybackpagebuffer;
```

```
"recovery procedures - establish a recovery point,
recover, and discard a recovery point"

establishrp procedure;
begin if (status=open) & writtento then
"ensure disc copy represents current file state"
IO.writedisc(filemap[activepageno],pagebuffer);
initialise-cache;
cache.oldfilestatus:=status; "checkpoint file status"
end establishrp;

recovery procedure;
var index : filemapindex; oldpage : discaddress;
begin "reset the necessary in-core variables"
if status = closed then openfile else activepageno:=0;
for each entry in cache.region do
  begin extract-from-entry(oldpage,index);
  E1.releasediscpage(filemap[index]); "throw new page away"
  filemap[index]:=oldpage; "reset filemap"
  end for loop;
if cache.oldfilestatus = closed then closefile;
tidyup-cache;
end recovery;

discardrp procedure;
var index : filemapindex; oldpage : discaddress;
begin for each entry in cache.region do
  begin extract-from-entry(oldpage,index);
  E1.releasediscpage(oldpage); "throw old page away"
  end for loop;
tidyup-cache;
end discardrp;
```

INCLUSIVE RECOVERY IN E2

```
"type and var declarations as before except for the cache"

type cacheentry = record oldpage : discaddress;
                         newpage : discaddress;
                  endrecord;

 ...

recoverycache var cache : record housekeepingvars : ...;
                                 region : array[1..P] of cacheentry;
                          endrecord;

entry procedure openfile; ...       "as before"
entry procedure closefile; ...      "as before"
entry procedure writeline(...); ... "as before"
entry procedure readline(...); ...  "as before"
procedure getpage(...); ...         "as before"

procedure copybackpagebuffer;
var newpage : discaddress;
begin if cacheing-required then "record recovery data"
  begin newpage:=E1.getdiscpage;
  "cache old and new disc page addresses"
  put-in-cache(filemap[index],newpage);
  filemap[index]:=newpage; "reset filemap"
  end;
IO.writedisc(filemap[index],pagebuffer);
end copybackpagebuffer;

establishrp procedure;
begin initialise-cache;
end establishrp;

recovery procedure;
var oldpage,newpage : discaddress;
begin for each entry in cache.region do
  begin extract-from-entry(oldpage,newpage);
  E1.releasediscpage(newpage); "throw new page away"
  end for loop;
tidyup-cache;
end recovery;

discardrp procedure;
var oldpage,newpage : discaddress
begin for each entry in cache.region do
   begin extract-from-entry(oldpage,newpage);
   E1.releasediscpage(oldpage); "throw old page away"
   end for loop;
 tidyup-cache;
 end discardrp;
```

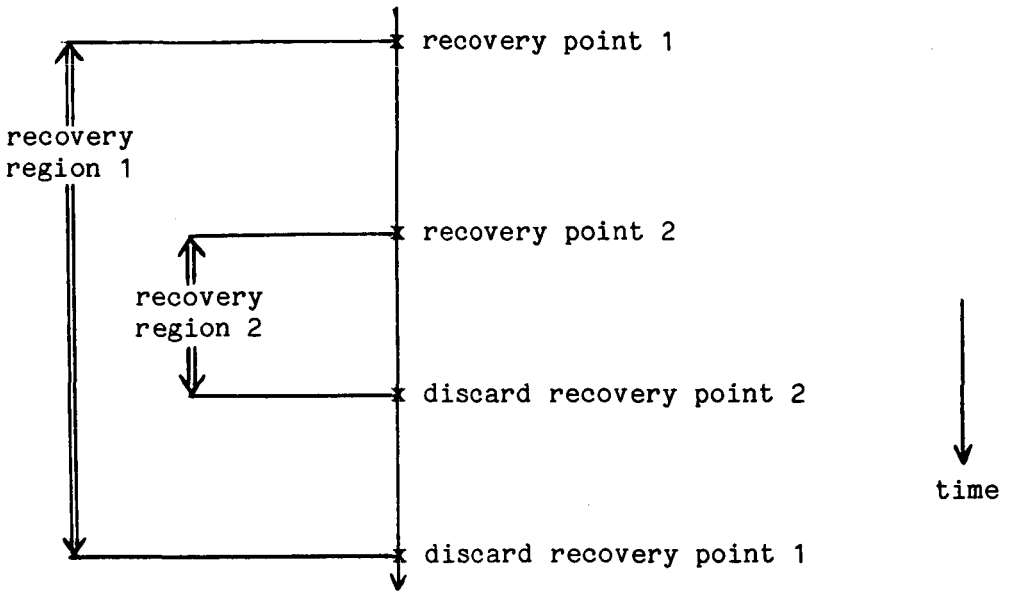

Figure 1. Multiple Recovery Points

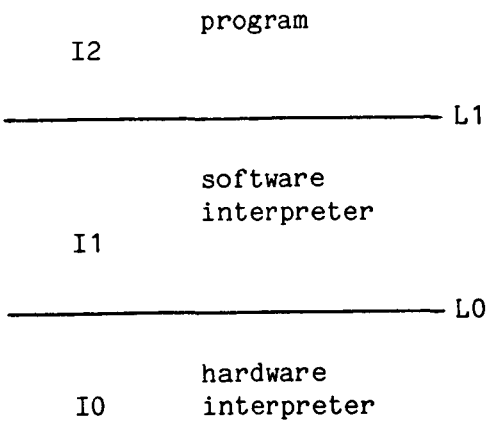

Figure 2. Interpretive Multilevel System

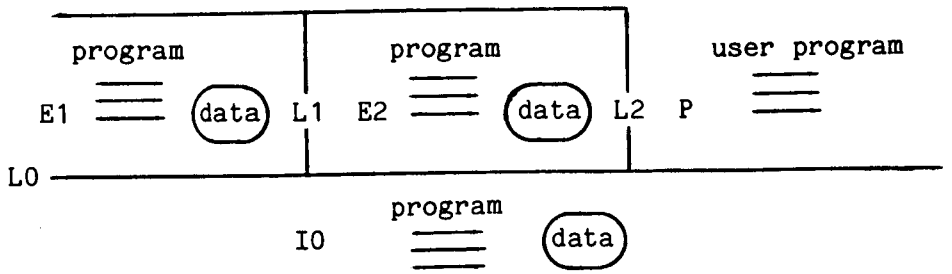

Figure 3. Extended Interpreter Multilevel System

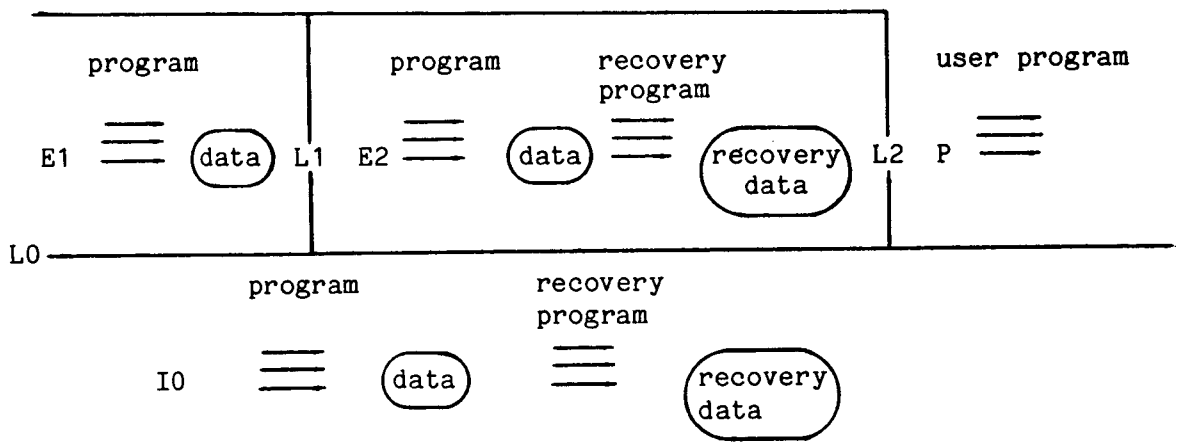

Figure 4. Recovery Structures in an Extended Interpreter System